Understanding the Timed Distributed Trace of a Partially Synchronous System at Runtime

Yiling Yang[1,2], Yu Huang[1,2*], Jiannong Cao[3], Jian Lu[1,2]
[1]State Key Laboratory for Novel Software Technology
Nanjing University, Nanjing 210046, China
[2]Institute of Computer Software
Nanjing University, Nanjing 210046, China
csylyang@gmail.com, {yuhuang, lj}@nju.edu.cn
[3]Internet and Mobile Computing Lab, Department of Computing
Hong Kong Polytechnic University, Hong Kong, China
csjcao@comp.polyu.edu.hk

***Abstract*—It has gained broad attention to understand the timed distributed trace of a cyber-physical system at runtime, which is often achieved by verifying properties over the observed trace of system execution. However, this verification is facing severe challenges. First, in realistic settings, the computing entities only have imperfectly synchronized clocks. A proper timing model is essential to the interpretation of the trace of system execution. Second, the specification should be able to express properties with real-time constraints despite the asynchrony, and the semantics should be interpreted over the currently-observed and continuously-growing trace. To address these challenges, we propose PARO - the *partially synchronous system observation* framework, which i) adopts the partially synchronous model of time, and introduces the lattice and the timed automata theories to model the trace of system execution; ii) adopts a tailored subset of TCTL to specify temporal properties, and defines the 3-valued semantics to interpret the properties over the currently-observed finite trace; iii) constructs the timed automaton corresponding to the trace at runtime, and reduces the satisfaction of the 3-valued semantics over finite traces to that of the classical boolean semantics over infinite traces. PARO is implemented over MIPA - the open-source middleware we developed. Performance measurements show the cost-effectiveness of PARO in different settings of key environmental factors.**

I. Introduction

Advances in sensor and actuator technologies have given rise to the influx of an increasing number of mobile robots with sensing and controlling abilities, besides the basic abilities of computation and communication. Distributed systems formed by the interconnections of such mobile robots enable a variety of novel applications. A typical example of such distributed systems is a group of mobile robots for collaborative tasks, e.g., patrolling a chemical plant [1], [2], [3].

Though the systems of collaborating mobile robots enable various novel applications, they are notoriously difficult to program. Each mobile robot executes a program implementing one or more distributed algorithms, moves and manipulates its environment, and exchanges messages over wireless communication channels with other robots. Even if the high-level distributed algorithms for these systems are well-understood, failures, timing-errors, and message delays make their implementation challenging. Moreover, the system of mobile robots must be aware of and adaptive to the situation they operate in. For example, the robots must know the relative locations of robots nearby, to collaboratively form some geometric pattern.

*Corresponding author.

The challenges above motivate us to observe and analyze the execution of a distributed system of mobile robots at runtime. By understanding the system execution, the system developer can delineate and detect symptoms of software bugs. The system itself can also achieve awareness of its situation to adapt its behavior accordingly. Runtime observation and analysis of system execution is an important technique to improve system accountability, and is a complement to the traditional design time techniques such as model checking [4].

Runtime observation and analysis of system execution is achieved by runtime verifying specified properties over the system execution trace. For example, the robots are designed to satisfy the property C_1: *all the robots should gather at the assembly point within 15 seconds*. By runtime verifying the property over the collected execution trace of the robots, we can know whether the property is satisfied or violated. The results of the verification can further provide useful guidance to the developer (e.g., for the debugging of the system) or the system itself (e.g., for triggering the runtime adaptation of the system). However, runtime verification over the system execution trace is facing severe challenges.

The primary challenge is the intrinsic asynchrony of the system. The robots do not share the same notion of time and communications among them suffer from uncertain delay [5], [2], [6], [7]. The challenge of the asynchrony becomes more severe due to the resource constraints on the mobile robots, the unreliability of the wireless communications, and the interaction with the physical environment. To cope with this challenge, a proper timing model fitting the actual system is essential to modeling the observed trace of the system execution.

The *synchronous model* can shield the underlying asynchrony of the system and provide a total-order illusion of all the events in the trace. However, the synchronous model is overly-optimistic in that the local clocks of the robots are never perfectly synchronized in realistic settings [2], [5]. Reasoning properties directly over the trace assumed to be perfectly timestamped may lead to the neglect of the potential violation of the properties [7].

The *asynchronous model* is the most general model since it makes no synchrony assumptions about the underlying system [8]. However, the asynchronous model is overly-pessimistic in that the existing synchrony of the system (e.g., the synchronized, though not perfect, clocks) is completely abandoned. The cost of reasoning over the asynchronous model is often prohibitively high [9], [10]. Moreover, the asynchronous model can only describe the temporal happen-before relation between events in the trace. Properties with real-time constraints (e.g., "within 15 seconds" in C_1) cannot be reasoned over the asynchronous model.

Varying from the synchronous model and the asynchronous model, the *partially synchronous model* can appropriately model at various levels the synchrony which reasonably exists in a realistic distributed system, such as the imperfectly synchronized clocks in our motivating scenario. Therefore, the cost of reasoning can be effectively restricted, unlike that of the asynchronous model [6]. Besides, we can also reason properties with real-time constraints with the knowledge that the trace is imperfectly timestamped, although techniques dedicated for handling the uncertainty resulting from the partially synchronous model are still in demand.

Another important challenge arises in the specification of system properties of interest to the specifier (i.e., the system developer or the system itself). The specifier usually has real-time requirements, e.g., in property C_1. To check these requirements over the execution trace, we are thus concerned with properties with metric-time constraints. To cope with this challenge, we should provide a formal specification mechanism which can express metric-time properties. Meanwhile, the partially synchronous model has the branching-time structure due to the uncertainty resulting from the asynchrony, i.e., we can derive multiple possible executions of the partially synchronous system, besides the actually observed one. Thus, the specification should also be able to capture the notion of branching time. In addition, as the system executes, the observed trace is continuously "growing" to a potentially infinite size. The specification should be interpreted over the currently-observed (but still growing) finite trace.

Discussions above necessitate a systematic scheme for formal specification and runtime verification of properties with metric-time constraints over the execution trace of a partially synchronous system. Toward this objective, we propose PARO - the *partially synchronous system observation* framework, which consists of three essential parts:

1) *Modeling of the Trace of System Execution.* We adopt the partially synchronous model of time. The lattice theory is employed to model the branching structure of the system execution trace, and the timed automata theory is employed to model the metric structure of the trace. Hence, we model the system execution trace as a continuously-growing timed automaton;
2) *Specification of Temporal Properties.* TCTL is adopted to specify temporal properties over the execution trace. TCTL has the branching time structure to cope with the asynchrony in the trace. It can also express the metric-time properties of concern to the specifier. We employ a tailored subset of TCTL to trade certain expressiveness for the efficiency of verification. We also define the 3-valued semantics for the TCTL formulas to be interpreted over the currently-observed finite trace;
3) *Verification of the Specified Property at Runtime.* We first construct the continuously-growing timed automaton corresponding to the currently-observed trace in an incremental way. Then, we reduce the satisfaction of the 3-valued semantics over the finite trace to that of the classical boolean semantics over the infinite trace.

The PARO framework is implemented and evaluated over MIPA - the open-source middleware we developed [11], [3]. The performance measurements show the cost-effectiveness of PARO in different settings of key environmental factors. A case study of the realistic mobile robot gathering scenario is also conducted to demonstrate the effectiveness of PARO (as detailed in Appendix A).

The rest of this paper is organized as follows. In Section II, III, and IV, we discuss the three essential parts of the PARO framework. In Section V, we present the implementation and performance measurements. In Section VI, we review the existing work. Section VII concludes the paper with a brief summary and the future work.

II. Modeling of the Trace of System Execution

Understanding the execution of a distributed system of mobile robots is achieved by verifying specified temporal properties over the trace of system execution [12]. A distributed system consists of a collection of processes $P^{(1)}, P^{(2)}, \cdots, P^{(n)}$. Examples of the processes include a software process manipulating a mobile robot. One checker process P_{che} is in charge of collecting the execution trace of the processes and verifying the specified property. In this section, we first discuss the partially synchronous model employed to interpret the trace of system execution. Then we discuss the modeling of the branching structure and the metric structure of the trace with the lattice theory and the timed automata theory, respectively.

A. The Partially Synchronous Model

In realistic settings, each process $P^{(k)}$ may have a local clock, and they synchronize their local clocks with an external *source clock* T. The external clock synchronization is widely adopted and is especially useful in loosely-coupled networks [13], e.g., the NTP protocol is used for external synchronization of the Internet [14].

We model the processes as a partially synchronous system with approximately-synchronized real-time clocks towards the external source clock. We assume a bound ε on the difference between local clocks and the source clock.[1] That is, for each event e with local clock timestamp t, the global time (referring to the source clock) is bounded by a time interval $I(e) = [lo, hi]$ with $lo = t - \varepsilon$ and $hi = t + \varepsilon$.[2] Note that we assume the time intervals of the events of the same process are non-decreasing and are consistent with the process order, and there

[1]Note that our framework allows ε to vary over time, and here we assume a fixed bound ε for the ease of interpretation.

[2]We assume that the system starts at time 0 for simplicity.

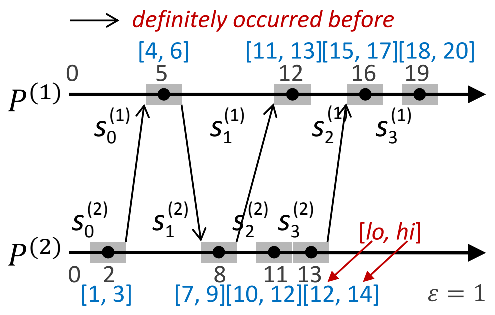

Fig. 1. Space time diagram

are various ways to ensure this assumption [6]. In the following sections, unless explicitly stated, we use lo and hi to denote the lower and upper bounds of a time interval, respectively.

B. Modeling the Branching Structure of Time

We first model the branching structure of time by the lattice of consistent global states (i.e., lattice of system snapshots). Then we define the possible temporal evolutions of the system state by the active-surface-induced CGS sequences.

1) Lattice of Consistent Global States: As the system executes, each process $P^{(k)}$ generates its (potentially infinite) trace of *local states* connected by *events*:"$e_0^{(k)}, s_0^{(k)}, e_1^{(k)}, s_1^{(k)}, \cdots$". Each local state $s_i^{(k)}$ is defined by the two adjacent events $e_i^{(k)}$ and $e_{i+1}^{(k)}$, denoted by $le(s_i^{(k)})$ and $he(s_i^{(k)})$, respectively. According to our timing model, the global time of each event can be bounded by a time interval $[lo, hi]$. Thus we can define the "*definitely occurred before*" relation (denoted by '$\rightarrow$') between the events and the states [6]. For two events e_1 and e_2, $e_1 \rightarrow e_2$ if i) they are on the same process and $e_1.lo < e_2.lo$; or ii) they are on different processes and $e_1.hi < e_2.lo$. Based on the relation between the events, we can further define the relation between local states. For two local states s_1 and s_2, $s_1 \rightarrow s_2$ if i) they are on the same process and $le(s_1) \rightarrow le(s_2)$; or ii) they are on different processes and $he(s_1) \rightarrow le(s_2)$. As shown in Fig. 1, the events (the black dots) are labeled with time intervals and the '$\rightarrow$' relation between the events is explicitly depicted.

A *global state* $\mathcal{G} = (s^{(1)}, s^{(2)}, \cdots, s^{(n)})$ is an n-tuple of local states from each $P^{(k)}$. Intuitively, a global state is consistent if an omniscient external observer could possibly observe that the system enters this state. Formally, a global state $\mathcal{C}$ is consistent iff the constituent local states are pairwise concurrent, i.e.,

$$\mathcal{C} = (s^{(1)}, s^{(2)}, \cdots, s^{(n)}), \forall\, i, j : i \neq j :: \neg(s^{(i)} \rightarrow s^{(j)})$$

A Consistent Global State (CGS) denotes a snapshot or a meaningful observation of the distributed system [15], [16]. We use $\mathcal{C}[k]$ to denote the k^{th} constituent local state of CGS $\mathcal{C}$. One key notion is that the set of observed CGSs has the lattice structure (denoted by LAT) [6]. The significance of time is that it restricts the possible interleavings of local states, which in turn determines the lattice of system snapshots. As the system executes, the observed lattice "grows" at runtime, to a potentially infinite size. Fig. 2 shows a currently-observed

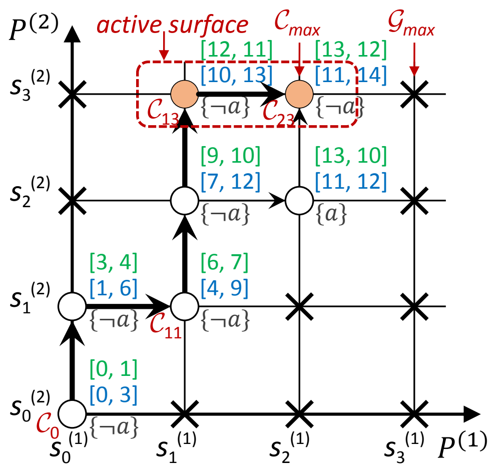

Fig. 2. Lattice of CGSs

lattice of Fig. 1. The dots '○' denote the CGSs and crosses '×' denote inconsistent global states.

We specify predicates over the CGSs to delineate properties concerning specific snapshot of the system. The predicates over CGSs can be viewed as the labeling of CGSs with letters from a finite alphabet AP (i.e., all pre-defined CGS predicates). Fig. 2 is an example where each CGS is labeled with the predicates ('a' or '$\neg a$') it satisfies.

2) Active-surface-induced CGS Sequences: As the system executes, the lattice "grows" and new CGSs will be added to *LAT* as successors of the *active surface CGSs* [9]. CGSs whose immediate successors (consistent or inconsistent global states) are not all discovered could have new immediate successors. We define these CGSs as the *active surface*. To formally define the active surface, first note that P_{che} uses a queue $Que^{(k)}(1 \leq k \leq n)$ to store the local states (in FIFO manner) sent from each $P^{(k)}$ [17], [18], [9]. Let $\mathcal{G}_{max}$ be the maximal observed global state (not necessarily consistent). The active surface is defined as:

$$Act(LAT) = \{\mathcal{C} \mid \mathcal{C} \in LAT, \exists k, \mathcal{C}[k] = \mathcal{G}_{max}[k]\}$$

We are concerned with the active surface, because when P_{che} observes a new local state from some process $P^{(k)}$, new CGSs stem from the active surface [9]. An example of the active surface is shown in Fig. 2.

In order to model the temporal evolution of the system state, we define the *CGS sequence* $Seq(\mathcal{C}_i, \mathcal{C}_j)$ as a sequence of CGSs. The bold line in Fig. 2 denotes a CGS sequence. We use $Seq[k]$ to denote the $(k+1)^{st}$ CGS of the CGS sequence Seq. Active-surface-induced CGS sequences, i.e., CGS sequences which originate from the initial CGS and span to the active surface CGSs, capture all possible temporal evolutions of the system state resulting from the asynchrony [9]. They are observed finite prefixes of the potentially infinite system state evolutions. Active-surface-induced CGS sequences of LAT are defined as:

$$Path(LAT) = \{Seq(\mathcal{C}_0, \mathcal{C}_i) \mid \mathcal{C}_i \in Act(LAT)\}$$

For example, in Fig. 2, all possible temporal evolutions of the system state are CGS sequences starting from $\mathcal{C}_0$ and currently ending at the active surface CGSs $\{\mathcal{C}_{13}, \mathcal{C}_{23}\}$.

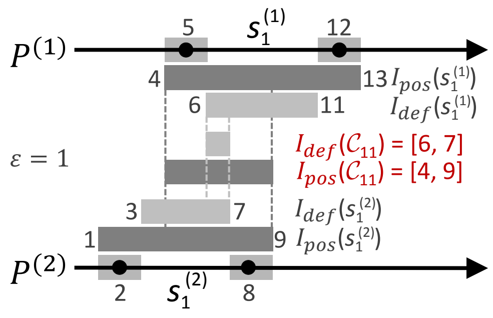

Fig. 3. Definite and possible intervals of a CGS

Please refer to our previous work for more detailed discussions on the modeling of asynchronous computations [9], [18], [10], [19].

C. Modeling the Metric Structure of Time

In this section, we discuss the modeling of the metric structure of time, based on the lattice structure discussed above. We first discuss the modeling of time information of the CGSs of the lattice, and then discuss the modeling of the lattice as a timed automaton.

1) Time Information of the CGSs: As the time of each event e is bounded by a global time interval $[lo, hi]$ (as shown in Fig. 1) and each local state s is defined by its adjacent events $le(s)$ and $he(s)$, we can define the *definite interval* and *possible interval* of a local state. The definite interval $I_{def}(s)$ of local state s is

$$I_{def}(s) = [le(s).hi, he(s).lo]$$

indicating that the process is definitely in local state s when the global time is in $I_{def}(s)$. Similarly, the possible interval $I_{pos}(s)$ of local state s is

$$I_{pos}(s) = [le(s).lo, he(s).hi]$$

indicating that the process is possibly in local state s when the global time is in $I_{pos}(s)$. As in Fig. 3, $I_{def}(s_1^{(1)}) = [6, 11]$ and $I_{pos}(s_1^{(1)}) = [4, 13]$ with $\varepsilon = 1$.

Notice that a CGS is a vector of local states from each process. We can further define the *definite interval* and *possible interval* of a CGS. The definite interval $I_{def}(\mathcal{C})$ of CGS $\mathcal{C}$ is the intersection of all the definite intervals of the constituent local states, i.e.,

$$I_{def}(\mathcal{C}) = [\max_{1\le k\le n} I_{def}(\mathcal{C}[k]).lo, \min_{1\le k\le n} I_{def}(\mathcal{C}[k]).hi]$$

indicating that the system is definitely in CGS $\mathcal{C}$ when the global time is in $I_{def}(\mathcal{C})$ (when $I_{def}(\mathcal{C}).lo \le I_{def}(\mathcal{C}).hi$). Similarly, the possible interval $I_{pos}(\mathcal{C})$ is the intersection of all the possible intervals of the constituent local states, i.e.,

$$I_{pos}(\mathcal{C}) = [\max_{1\le k\le n} I_{pos}(\mathcal{C}[k]).lo, \min_{1\le k\le n} I_{pos}(\mathcal{C}[k]).hi]$$

indicating that the system is possibly in CGS $\mathcal{C}$ when the global time is in $I_{pos}(\mathcal{C})$. Take the CGS $\mathcal{C}_{11}$ in Fig. 2 as an example, $I_{def}(\mathcal{C}_{11}) = [6, 7]$ and $I_{pos}(\mathcal{C}_{11}) = [4, 9]$, as shown in Fig. 3. Each CGS of the lattice in Fig. 2 is equipped with time intervals, with the upper one indicating the definite interval and the lower one indicating the possible interval.

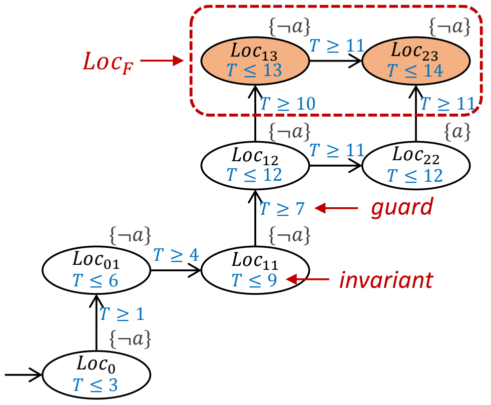

Fig. 4. Timed automaton $TA(LAT)$

2) Lattice as a Timed Automaton: The continuously-growing lattice equipped with time intervals can be characterized by a continuously-growing *timed automaton* adapted from the classical timed automaton [20], [4]. The timed automaton corresponding to a currently-observed lattice is a tuple:

$$TA(LAT) = (Loc, T, \hookrightarrow, Loc_0, Inv, AP, L, Loc_F)$$

where

- Loc is a set of locations, i.e., the set of CGSs of LAT;
- T is the global clock;
- $\hookrightarrow \subseteq Loc \times CC(T) \times Loc$ is a transition relation;
- $Loc_0 \in Loc$ is the initial location, i.e., the initial CGS $\mathcal{C}_0$ of LAT;
- $Inv : Loc \mapsto CC(T)$ is an invariant-assignment function;
- AP is a finite set of all pre-defined CGS predicates;
- $L : Loc \mapsto 2^{AP}$ is a labeling function for the locations;
- $Loc_F \subseteq Loc$ is a finite set of accepting locations, i.e., the active surface CGSs $Act(LAT)$.

Here, $CC(T)$ denotes the set of clock constraints over T in the form

$$g ::= T < c \mid T \le c \mid T > c \mid T \ge c \mid g \wedge g \ (c \in \mathbb{N})$$

The transformation from the lattice in Fig. 2 to a timed automaton is illustrated in Fig. 4. The locations of the timed automaton correspond to the CGSs of the lattice. The *invariant* of each location $\mathcal{C}$ is in the form $T \le I_{pos}(\mathcal{C}).hi$, indicating the time that the system can stay at $\mathcal{C}$. The *guard* of each transition to a location $\mathcal{C}$ is in the form $T \ge I_{pos}(\mathcal{C}).lo$, indicating when the transition can be taken.

The finite paths accepted by the timed automaton can be represented by a transition system $TS(TA(LAT))$ (or $TS(TA)$ for short) [4]. The states S in $TS(TA)$ are defined as $S = \langle \mathcal{C}, t\rangle$, where $\mathcal{C}$ is the current location and t is the current value of the clock T. The finite paths $Path_{fin}(TS(TA))$ of $TS(TA)$ are the active-surface-induced CGS sequences equipped with timestamps, in the form

$$\begin{aligned}\pi = \ & \langle Seq[0], 0\rangle \xrightarrow{d_1} \langle Seq[0], d_1\rangle \xrightarrow{e} \langle Seq[1], d_1\rangle \xrightarrow{d_2} \\ & \langle Seq[1], d_1+d_2\rangle \xrightarrow{d_3} \cdots \xrightarrow{d_j} \langle Seq[i], d_1+\cdots+d_j\rangle\end{aligned}$$

with $Seq(\mathcal{C}_0, \mathcal{C}_i) \in Path(LAT)$, $Seq[i] \in Loc_F$, and

$$I_{def}(Seq[i]).hi \leq d_1 + \cdots + d_j \leq I_{pos}(Seq[i]).hi$$ [3]

We use $\pi[k]$ to denote the $(k+1)^{st}$ state of path π, use $\pi[k].\mathcal{C}$ to denote the location of $\pi[k]$, and use $\pi[k].t$ to denote the time value of $\pi[k]$. Notice that each path starts with the initial state $\langle Loc_0, 0\rangle$, and continuously grows as the lattice grows, to a potentially infinite size.

III. Specification of Temporal Properties

Our specification inherits the notions of branching time and metric time from TCTL [20]. It is a tailored subset of TCTL, which trades certain expressiveness for the efficiency of verification. The 3-valued semantics is adopted to cope with the verification over the currently-observed finite trace, in contrast to the traditional boolean semantics over the infinite trace of possible system execution. We first present the syntax and then discuss the 3-valued semantics.

A. Syntax

The syntax of our specification is defined as follows:

$$\begin{aligned} \Phi &::= \exists\varphi \mid \forall\varphi \\ \varphi &::= \Diamond^J \phi \mid \Box^J \phi \\ \phi &::= a \mid g \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi \mid \phi \Rightarrow \phi \end{aligned}$$

where $a \in AP$, $g \in CC(T)$, and $J \subseteq \mathbb{R}_{\geq 0}$ is a time interval bounded by natural numbers [4].

Our specification is a subset of TCTL without nested modal operators [21]. It can express numerous properties in our scenario. Informally, $\exists\Diamond^J\phi$ means ϕ possibly holds during the interval J, $\exists\Box^J\phi$ means Φ potentially always holds in J, $\forall\Diamond^J\phi$ means ϕ always eventually holds in J, and $\forall\Box^J\phi$ means ϕ invariantly holds in J. Notice that $\exists\Box^J\phi = \neg\forall\Diamond^J\neg\phi$ and $\forall\Box^J\phi = \neg\exists\Diamond^J\neg\phi$. For example, the property C_1 (i.e., "*all the robots should gather at the assembly point within 15 seconds*") in Section I can be expressed as $\Phi_{C_1} = \forall\Diamond^{[0,15]}a$ *with* a = "*all the robots are at the assembly point*".

B. The 3-valued Semantics

We first discuss the classical boolean semantics over infinite paths, then discuss why the 3-valued semantics is inevitable for finite paths, and finally present the 3-valued semantics.

Given an infinite path $\widetilde{\pi} \in Path_{inf}(\widetilde{TS})$ of a transition system $\widetilde{TS}$ with infinite paths and a time $j \geq 0$, we can get the locations in the path at time j, denoted by

$$Loc(\widetilde{\pi}, j) = \{\widetilde{\pi}[k].\mathcal{C} \mid k \geq 0 \wedge \widetilde{\pi}[k].t \leq j \wedge \widetilde{\pi}[k+1].t \geq j\}$$

Note that we can easily check that whether a location $\mathcal{C}$ satisfies a predicate ϕ, i.e., $\mathcal{C} \models \phi$.[4] Thus, we can check whether an infinite path $\widetilde{\pi}$ satisfies a path formula φ:

$$\widetilde{\pi} \models \Diamond^J\phi \quad \text{iff} \quad \exists j \in J, \exists\mathcal{C} \in Loc(\widetilde{\pi}, j), \mathcal{C} \models \phi \tag{1}$$

$$\widetilde{\pi} \models \Box^J\phi \quad \text{iff} \quad \forall j \in J, \forall\mathcal{C} \in Loc(\widetilde{\pi}, j), \mathcal{C} \models \phi \tag{2}$$

[3]The lower bound of $d_1 + \cdots + d_j$ can be tightly bounded by the value in Eq. (11) by replacing $\mathcal{C}$ with $Seq[i]$.

[4]The details are omitted here for brevity.

Then we can easily check whether a transition system $\widetilde{TS}$ with infinite paths satisfies a TCTL formula Φ:

$$\widetilde{TS} \models \exists\varphi \quad \text{iff} \quad \exists\widetilde{\pi} \in Path_{inf}(\widetilde{TS}), \widetilde{\pi} \models \varphi \tag{3}$$

$$\widetilde{TS} \models \forall\varphi \quad \text{iff} \quad \forall\widetilde{\pi} \in Path_{inf}(\widetilde{TS}), \widetilde{\pi} \models \varphi \tag{4}$$

The timed automaton $\widetilde{TA}$ of the transition system $\widetilde{TS}$ satisfies a TCTL formula Φ iff the transition system $\widetilde{TS}$ satisfies the formula Φ [4], i.e.,

$$\widetilde{TA} \models \Phi \quad \text{iff} \quad \widetilde{TS}(\widetilde{TA}) \models \Phi \tag{5}$$

However, the paths of the transition system corresponding to the lattice are finite, and finite paths may not be sufficient to either satisfy or falsify TCTL formulas [22], [10]. For example, based on the classical boolean semantics of TCTL, the timed automaton in Fig. 4 does not satisfy the formula Φ_{C_1}. The timed automaton grows as the lattice grows, and there may be a new successor location of Loc_{13} and Loc_{23} satisfying 'a' (as in Fig. 5), which may lead to the satisfaction of Φ_{C_1}. That is to say, the classical semantics of TCTL does not provide intuitive and convenient support for this case of "being inconclusive" over the currently-observed finite trace. This case of being inconclusive may often appear when verifying TCTL formulas over the observed finite trace at runtime.

Discussions above motivate us to adopt the 3-valued semantics, i.e., providing a third value "inconclusive" for the case of being inconclusive [22], [10]. We use the symbols '$\top$', '$\bot$', and '?' to denote "*true*", "*false*", and "*inconclusive*", respectively. Let $\widetilde{\Pi}$ be the set of all the infinite timed paths with time non-decreasing. The semantics of whether a finite path π satisfies a path formula φ is defined as follows:

$$[\pi \models \varphi] = \begin{cases} \top & \text{if } \forall\sigma, \pi\sigma \in \widetilde{\Pi}, \pi\sigma \models \varphi \\ \bot & \text{if } \forall\sigma, \pi\sigma \in \widetilde{\Pi}, \pi\sigma \not\models \varphi \\ ? & \text{otherwise.} \end{cases} \tag{6}$$

The semantics of our specification is defined as follows:

$$\begin{aligned} &[TS(TA) \models \exists\varphi] \\ &= \begin{cases} \top & \text{if } \exists\pi \in Path_{fin}(TS(TA)), [\pi \models \varphi] = \top \\ \bot & \text{if } \forall\pi \in Path_{fin}(TS(TA)), [\pi \models \varphi] = \bot \\ ? & \text{otherwise.} \end{cases} \end{aligned} \tag{7}$$

$$\begin{aligned} &[TS(TA) \models \forall\varphi] \\ &= \begin{cases} \top & \text{if } \forall\pi \in Path_{fin}(TS(TA)), [\pi \models \varphi] = \top \\ \bot & \text{if } \exists\pi \in Path_{fin}(TS(TA)), [\pi \models \varphi] = \bot \\ ? & \text{otherwise.} \end{cases} \end{aligned} \tag{8}$$

$$[TA \models \Phi] = \begin{cases} \top & \text{iff } [TS(TA) \models \Phi] = \top \\ \bot & \text{iff } [TS(TA) \models \Phi] = \bot \\ ? & \text{otherwise.} \end{cases} \tag{9}$$

IV. Verification of the Specified Property at Runtime

In this section, we discuss the verification of the specified property at runtime. Each time a new local state of some process is sent to P_{che}, P_{che} first incrementally updates the new active surface, constructs the corresponding timed automaton, and then checks the property Φ over the timed automaton. The checking of Φ over the timed automaton is achieved by checking Φ over two special extended timed automata with infinite paths.

We first discuss the incremental maintenance of the active surface and the corresponding timed automaton, and then discuss the verification of the specified property.

A. Maintenance of the Timed Automaton

P_{che} continuously collects the execution trace from the processes, and maintains the active surface (not the whole lattice of system snapshots) at runtime. The maintenance of the active surface is incremental in that new CGSs can grow from the active surface. With the notion of the active surface, evolution of the lattice can be viewed as discarding the "old" nodes in the active surface and obtaining the "new" ones incrementally at runtime [9]. The worst-case number of active surface CGSs is in $O(np^{n-1})$, where p is the upper bound of the number of local states of each process, and n is the number of processes. (Note that the worst-case number of CGSs of the whole lattice is $O(p^n)$.) Please refer to our previous work [9] for more detailed discussions on the runtime maintenance algorithm of the active surface.

Based on the incremental maintenance of the active surface, the timed automaton corresponding to the lattice can also be incrementally constructed. Notice that each location of the timed automaton corresponds to one CGS of the lattice. The successor CGSs of the inactive CGSs (e.g., the white CGSs in Fig. 2) have already been discovered, and the corresponding part of the timed automaton (e.g., the white locations in Fig. 4) also has been completely constructed. Thus, the timed automaton can be incrementally constructed based on the runtime maintenance of the active surface. Whenever a new active CGS is added to the active surface, a new location corresponding to the new CGS is added to the timed automaton.

B. Verification of the Specified Property

Though our specification is a tailored subset of TCTL, we cannot directly apply the standard TCTL model checking algorithms on the timed automaton TA due to the 3-valued semantics dedicated for finite paths of $TS(TA)$. According to the 3-valued semantics in Eq. (6), we should append each finite path π with all possible infinite suffixes σ and check whether the infinite paths $\pi\sigma \in \widetilde{\Pi}$ satisfy the path formula φ. Rather than appending the finite path with all possible infinite suffixes, we define two special types of infinite suffixes $\sigma_\top$ and $\sigma_\bot$ with $\forall k, \sigma_\top[k].\mathcal{C} \models \phi$ and $\sigma_\bot[k].\mathcal{C} \not\models \phi$, to ease the verification. Then, Eq. (6) can be rewritten as follows:

$$[\pi \models \varphi] = \begin{cases} \top & \text{if } \pi\sigma_\top \models \varphi \wedge \pi\sigma_\bot \models \varphi \\ \bot & \text{if } \pi\sigma_\top \not\models \varphi \wedge \pi\sigma_\bot \not\models \varphi \\ ? & \text{otherwise.} \end{cases} \quad (10)$$

Based on the semantics above, we can extend each path $\pi \in Path_{fin}(TS(TA))$ with the two types of infinite suffixes $\sigma_\top$ and $\sigma_\bot$. The extension of $\sigma_\top$ is achieved by adding an extra location Loc_{inf} with $L(Loc_{inf}) = \{\phi\}$ to the timed automaton, and adding transitions from each of the locations $\mathcal{C} \in Loc_F$ to Loc_{inf} with the guard

$$T \geq \min_{i \in \{k | \mathcal{C}[k] = \mathcal{G}_{max}[k]\}} \max(I_{pos}(\mathcal{C}).lo, I_{def}(\mathcal{C}[i]).hi) \quad (11)$$

The guard can be understood by assuming that, in dimension i of the CGS $\mathcal{C}$, a successor CGS $\mathcal{C}'$ is coming. The guard from $\mathcal{C}$ to $\mathcal{C}'$ is $T \geq I_{pos}(\mathcal{C}').lo = \max(I_{pos}(\mathcal{C}).lo, I_{pos}(\mathcal{C}'[i]).lo)$ with $I_{pos}(\mathcal{C}'[i]).lo = I_{def}(\mathcal{C}[i]).hi$. The extension of $\sigma_\bot$ is in the same way except that $L(Loc_{inf}) = \{\neg\phi\}$. After that, we can get two extended timed automata $\widetilde{TA}_\top$ and $\widetilde{TA}_\bot$ with infinite paths. The extensions to the timed automaton of Fig. 4 are shown in Fig. 5 and Fig. 6. The location Loc_{inf} in Fig. 5 is labeled with $\{a\}$ and the location Loc_{inf} in Fig. 6 is labeled with $\{\neg a\}$. In Fig. 5, based on Eq. (11), the guard of the transition from Loc_{13} to Loc_{inf} is $T \geq \max(I_{pos}(\mathcal{C}_{13}).lo, I_{def}(\mathcal{C}_{13}[2]).hi)$. Notice that $I_{pos}(\mathcal{C}_{13}).lo = 10$ (as shown in Fig. 2), and $I_{def}(\mathcal{C}_{13}[2]).hi = I_{def}(s_3^{(2)}).hi = 12$ (as shown in Fig. 1). Thus, the guard of the transition from Loc_{13} to Loc_{inf} is $T \geq 12$.

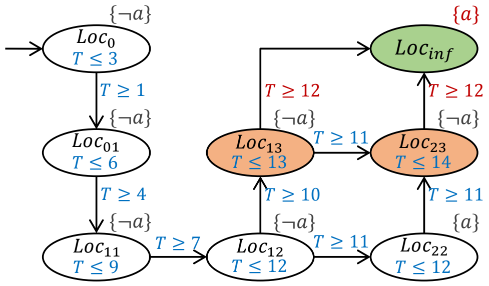

Fig. 5. Extended timed automaton $\widetilde{TA}_\top$

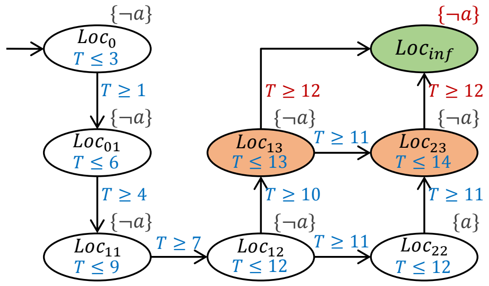

Fig. 6. Extended timed automaton $\widetilde{TA}_\bot$

Based on Eq. (3)-(5), we can check whether $\widetilde{TA}_\top \models \Phi$ and $\widetilde{TA}_\bot \models \Phi$. Then the semantics of our specification in Eq. (9) can be rewritten as follows:

$$[TA \models \Phi] = \begin{cases} \top & \text{if } [\widetilde{TA}_\top \models \Phi] = [\widetilde{TA}_\bot \models \Phi] = \top \\ \bot & \text{if } [\widetilde{TA}_\top \models \Phi] = [\widetilde{TA}_\bot \models \Phi] = \bot \\ ? & \text{otherwise.} \end{cases} \quad (12)$$

Consequently, the verification boils down to the verification of $\widetilde{TA}_\top \models \Phi$ and $\widetilde{TA}_\bot \models \Phi$. Based on the two timed automata in Fig. 5 and Fig. 6, we can know that $[TA \models \Phi_{C_1}] = ?$, i.e., the result is "inconclusive".

The verification of the formula Φ on the timed automaton $\widetilde{TA}$ is achieved by checking the derived CTL formula $\hat{\Phi}$ on $\widetilde{TA}$ [4]. The time interval J in the formula Φ is eliminated by adding equivalent clock constraints into the ϕ part. We can transform the TCTL formula Φ into a CTL formula $\hat{\Phi}$ as

Algorithm 1: Verification algorithm on P_{che}

```
1 Upon initialization
2     get property $\Phi$, and transfer $\Phi$ into CTL formula $\hat{\Phi}$;
3 Upon receiving local state $s_i^{(k)}$ from $P^{(k)}$
4     construct $Act(LAT)$ with $s_i^{(k)}$ incrementally;
5     construct $TA(LAT)$ incrementally;
6     extend $TA(LAT)$ into $\widetilde{TA}_\top$ and $\widetilde{TA}_\bot$;
7     check $\widetilde{TA}_\top \models_{CTL} \hat{\Phi}$ and $\widetilde{TA}_\bot \models_{CTL} \hat{\Phi}$;
8     check $TA \models \Phi$ according to Eq. (12)-(13);
```

follows[5]:

$$\begin{aligned} \Phi = \exists\varphi \quad &\text{then} \quad \hat{\Phi} = \exists\hat{\varphi} \\ \Phi = \forall\varphi \quad &\text{then} \quad \hat{\Phi} = \forall\hat{\varphi} \\ \varphi = \Diamond^J \phi \quad &\text{then} \quad \hat{\varphi} = \Diamond(T \in J \wedge \phi) \\ \varphi = \Box^J \phi \quad &\text{then} \quad \hat{\varphi} = \Box(T \in J \Rightarrow \phi) \end{aligned}$$

For example, the derived CTL formula of $\Phi_{C_1} = \forall\Diamond^{[0,15]}a$ is $\hat{\Phi}_{C_1} = \forall\Diamond(T \leq 15) \wedge a)$.

The equivalence of the satisfaction is ensured as follows:

$$\widetilde{TA} \models \Phi \quad \text{iff} \quad \widetilde{TA} \models_{CTL} \hat{\Phi} \tag{13}$$

The proof is straightforward and omitted here [4]. Thus, the verification is finally reduced to the verification of $\widetilde{TA}_\top \models_{CTL} \hat{\Phi}$ and $\widetilde{TA}_\bot \models_{CTL} \hat{\Phi}$, i.e., checking a CTL formula (without nested modal operators) on a classical timed automaton (with only one clock), which can be efficiently achieved by numerous optimization algorithms [4], [23], [24], [25]. The skeleton of the verification algorithm is shown in Algorithm 1.

V. Experiments

In this section, we first describe the implementation of PARO, and then discuss the performance evaluation. The effectiveness of PARO is demonstrated through a case study of a realistic mobile robot gathering scenario based on our RobotCar project. Details of the case study can be found in Appendix A.

A. Implementation

We implement PARO on the open-source middleware we developed - *Middleware Infrastructure for Predicate detection in Asynchronous environments* (MIPA) [11]. Based on MIPA, we specify properties in TCTL formulas (e.g., the formula Φ_{C_1}) to the middleware using an XML schema. The devices (e.g., mobile robots) register themselves to the middleware (abstracted as processes), and continuously send the trace with local timestamps to the checker processes on the middleware. Checker processes are implemented as third-party services on the middleware, in charge of collecting related traces and verifying the formulas. The verification of $\widetilde{TA}_\top \models_{CTL} \hat{\Phi}$ and $\widetilde{TA}_\bot \models_{CTL} \hat{\Phi}$ (line 7 of Algorithm 1) is achieved by incrementally generating XML descriptions for the timed automata and automatically invoking UPPAAL [21], which is a toolbox for verification of real-time systems. Each time a formula is verified "true", "false", or "inconclusive", the middleware will notify the users.

[5]As we do not allow nested modal operators in the specification and the timed automaton has no clock resets, there is no need for introducing a fresh clock to measure the elapse of time, as in [4].

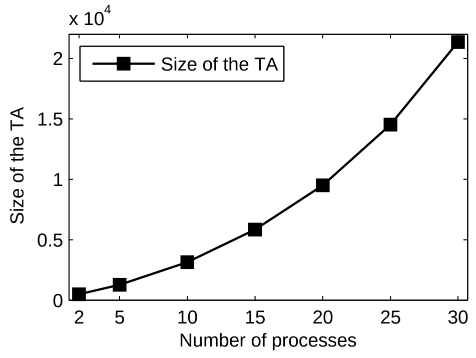

Fig. 7. The size of the TA vs. the number of processes (ε = 200 ms)

B. Performance Evaluation

In this section, we conduct simulations of the robot gathering scenario to evaluate the performance of PARO under different settings of key environmental factors. We first describe the experiment setup and then discuss the evaluation results.

We let the robots collect sensing data every second. We generate the sensing data using the Poisson distribution. Specifically, the average time of local activities (where the local predicate is true) on the robots is 10 s, and the average interval between the activities (where the local predicate is false) is 5 s. The number of the sensing data items on each robot is up to 2,400. The lifetime of the experiments is up to 40 mins. The experiments are conducted on a PC running Windows 8.1 (x64) and Java version 1.7 with an Intel Core i5-2400 Quad-Core Processor (3.10 GHz) and 8 GB of memory.

In the experiments, we check the formula $\forall\Diamond^{[0,15000]}(LP_1 \wedge \cdots \wedge LP_n)$[6] and tune two key environmental factors - the number of processes (i.e., the mobile robots) n and the clock difference bound ε - to evaluate the five performance metrics S_M, S_U, T_M, T_U, and $|Loc|$. S_M denotes the average memory cost of MIPA (mainly for the construction of the active surface of the lattice and the timed automaton), S_U denotes the average memory cost of UPPAAL, T_M denotes the average time cost for the construction of the active surface and the timed automaton by MIPA (line 4-6 of Algorithm 1), T_U denotes the average time cost for the verification by UPPAAL (line 7-8 of Algorithm 1), and $|Loc|$ denotes the size of the timed automaton TA when the experiment stops. $S_M + S_U$ indicates the average of the total memory cost, and $T_M + T_U$ indicates the average of the total latency.

1) Effects of Tuning the Number of Processes: In this experiment, we study how the number of processes n affects the performance of PARO. We fix the clock difference bound ε to 200 ms, and tune n from 2 to 30.

[6]The local predicate $LP_i = (R_i.front \leq 600\ mm \wedge R_i.left \leq 400\ mm)$, indicating robot R_i is at the assembly point.

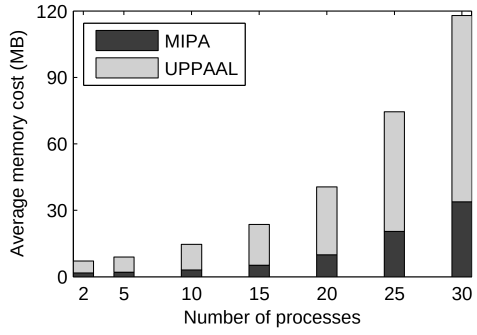

Fig. 8. Average memory cost vs. the number of processes (ε = 200 ms)

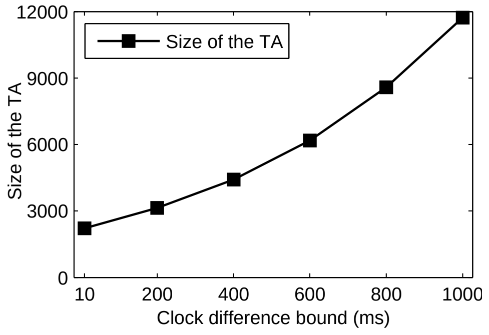

Fig. 10. The size of the TA vs. the clock difference bound ε (n = 10)

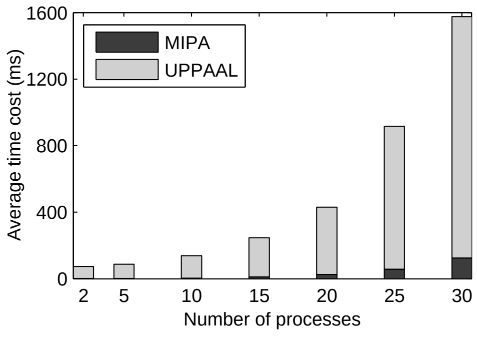

Fig. 9. Average time cost vs. the number of processes (ε = 200 ms)

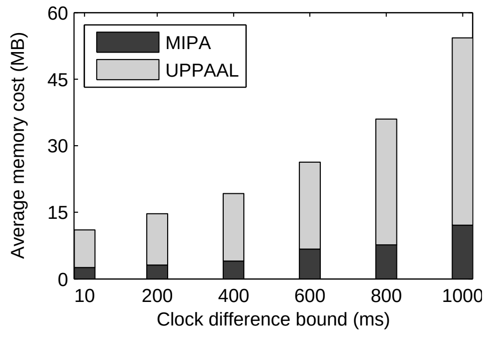

Fig. 11. Average memory cost vs. the clock difference bound ε (n = 10)

As shown in Fig. 7, the increase of n leads to fast increase in the size of the TA (from 484 to 21,354). Compared to the size of LAT under the asynchronous model in our previous work [9], the size of the lattice (i.e., $|Loc|$) in this work is rather smaller. To a certain extent, the lattice structure turns out to be applicable on the partially synchronous model, although it is exponential on the asynchronous model. The reason is that, we have made use of the existing synchrony of the system and efficiently restricted the size of the lattice, comparing to previous work under the asynchronous model [16], [9].

As shown in Fig. 8, the average memory cost of MIPA S_M increases quickly as n increases, while S_U (the average memory cost of UPPAAL) increases a little faster than S_M. As we tune n from 2 to 30, S_M and S_U increase from 1.8 MB to 34 MB and from 5.3 MB to 84 MB, respectively. That is to say, the memory cost of verification is larger than that of the construction of the active surface and the timed automaton. Moreover, the increase of $S_M + S_U$ is roughly in accordance with the increase of the size of the TA in Fig. 7. The reason is that our TCTL formulas have no nested modal operators, thus can be space-efficiently verified over the TA.

As for the time cost, the average time cost of MIPA T_M increases slowly from 1.8 ms to 125 ms as n increases from 2 to 30, as shown in Fig. 9. However, the average time cost of UPPAAL T_U increases quickly from 72 ms to 1,450 ms. Most of the time (over 90%) is spent on the verification. The average of the total latency is acceptable. Furthermore, the increase of T_U is roughly in accordance with the increase of the size of the TA in Fig. 7. That is to say, PARO is relatively time-efficient.

2) Effects of Tuning the Clock Difference Bound: In this experiment, we study how the clock difference bound ε affects the performance of PARO. We fix the number of processes n to 10, and tune ε from 10 ms to 1 s.

As shown in Fig. 10, the increase of ε leads to quick increase in the size of the TA (from 2,220 to 11,722). This is because the increase of ε leads to more possible interleavings of events, thus leads to more possible system snapshots. Compared to Fig. 7, the increase of the size of the TA caused by ε is slower than that caused by n, i.e., the number of processes n has greater impact on the size of the TA than the clock difference bound ε. As the bound of clock difference is often small in realistic systems, the number of processes n becomes the most important environmental factor.

As shown in Fig. 11, the average memory cost of MIPA S_M increases slowly as ε increases, while S_U (the average memory cost of UPPAAL) increases a little faster than S_M. As we tune ε from 10 ms to 1,000 ms, S_M and S_U increase from 2.6 MB to 12 MB and from 8.5 MB to 42 MB, respectively. The memory cost of verification by UPPAAL is larger than that of the construction of the active surface and the timed automaton by MIPA, as that in Fig. 8. Moreover, the increase of $S_M + S_U$ is roughly in accordance with the increase of the size of the TA in Fig. 10. The reason is the same as that for the number of processes. Consequently, PARO is relatively space-efficient.

As for the time cost, the average time cost of MIPA T_M increases slowly from 2.2 ms to 93 ms as ε increases from 10 ms to 1,000 ms, as shown in Fig. 12. However, the average time cost of UPPAAL T_U increases quickly from 78 ms to 1,222 ms. Most of the time (over 90%) is spent on the verification. The average of the total latency is acceptable. Notice that although T_U increases quickly, it is roughly in accordance with

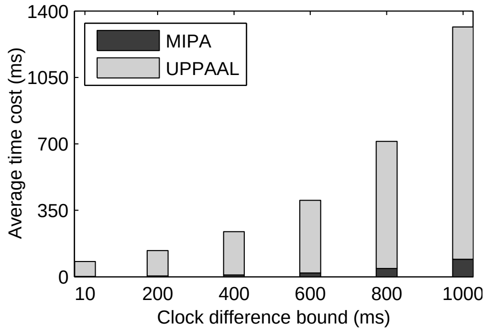

Fig. 12. Average time cost vs. the clock difference bound ε (n = 10)

the increase of the size of the TA in Fig. 10. That is to say, PARO is relatively time-efficient.

VI. Related Work

Our work can be posed against two areas of related work: model checking of real-time systems and detection of global predicates over distributed computations.

In the area of model checking of real-time systems, the existing work is mostly studied and developed in the framework of Alur and Dill's Timed Automata, and TCTL gains extensive research since then [4], [26], [20]. Most work focuses on finding efficient algorithms for timed automata verification [4], [23], [24], [25]. These optimization algorithms are generally orthogonal to PARO. Although PARO shares many similarities with model checking over timed automata, there are important differences. First, the complete model (e.g., a series of timed automata) of the system is mandatory in model checking over timed automata [4], [26], [20]. In contrast, as an external observer of an already-running system, our work is applicable to "black box" systems with no system model at hand. Model checking deals with infinite traces of all possible executions, whereas our work deals with the currently-observed finite trace of one concrete execution, by modeling the finite trace as a continuously-growing timed automaton. The temporal logics of model checking, such as TCTL, are interpreted over infinite traces, and the checking cost is often prohibitive [4]. However, to trade the expressiveness for the checking cost, our specification is a tailored subset of TCTL without nested modal operators, as in [21]. Moreover, unlike the classical semantics of TCTL over infinite traces, we adopt 3-valued semantics for our specification over the currently-observed finite trace to cope with the case of "being inconclusive".

In detection of global predicates over distributed computations (aka "runtime verification"), existing work can be categorized by the timing model. Based on the synchronous model, predicates can be detected over a single total order of events [22], [27], [28]. Bauer et al. [22] detect 3-valued semantics of LTL and TLTL over a sequence of timed events. Our 3-valued semantics of TCTL is partially inspired by this work [22]. Kshemkalyani [27] detects predicates concerning the relationships of time intervals over event streams. Xu et al. [28] detect first-order logic based formulas over collected contextual activities. However, the actual system is imperfectly synchronized in our scenario, which makes the work above inapplicable. Based on the asynchronous model, predicates are detected over multiple possible total orders of events consistent with the 'happen-before' relation resulting from message passing [12], [15], [16], [17], as in our previous work [29], [18], [9], [10]. We detect conjunctive predicates in [29], [18], regular expression predicates in [9], and 3-valued CTL predicates in [10]. However, the asynchronous model is overly-pessimistic in that the existing synchrony of the system is completely abandoned, and the detection over the asynchronous model is usually expensive [9], [10]. Based on the partially synchronous model, predicates can be detected at a relatively lower cost [30], [31], [6], [2]. Marzullo et al. [30], Mayo et al. [31], and Stoller [6] detect global predicates without timing constraints in partially synchronous systems. Duggirala et al. [2] detect whether there exists a real-time t, when bounded executions of the system that correspond to the trace satisfy a given global predicate. In the work [2], the complete system model as a timed input/output automaton is mandatory. In contrast, our PARO works over the trace with no system model at hand. We model the observed trace as a continuously-growing timed automaton, and check a subset of TCTL formulas with 3-valued semantics over the timed automaton.

VII. Conclusion and Future Work

In this work, we propose the PARO framework for formal specification and runtime verification of properties with metric-time constraints over the trace of a partially synchronous system of mobile robots. The PARO framework consists of three essential parts: 1) modeling of the trace of system execution; 2) specification of temporal properties; 3) verification of the specified property at runtime.

In our future work, we need to design an optimized incremental algorithm for the detection of $\widetilde{TA}_\top \models_{CTL} \hat{\Phi}$ and $\widetilde{TA}_\bot \models_{CTL} \hat{\Phi}$, since the automaton $TA(LAT)$ is incrementally constructed as the active surface of the lattice evolves. We also need to dig into the specific type of timed automaton corresponding to the timed trace, and study how to detect the full TCTL over the timed automaton while preserving a relatively low checking cost. In this work, we investigate partially synchronous systems with the processes synchronizing their clocks with an external source clock. In our future work, we need to investigate runtime verification of properties in partially synchronous systems with the processes synchronizing their clocks internally. A more comprehensive experimental evaluation is also necessary.

Acknowledgements

This work is supported by the National 973 Program of China (2015CB352202), the National Science Foundation of China (61272047, 91318301, 61321491), and the Program A for Outstanding PhD candidate of Nanjing University. We are grateful to Lei Bu's insightful comments and Chao Fang's implementations on the mobile robots.

References

[1] G. Zhan and W. Shi, "Lobot: Low-cost, self-contained localization of small-sized ground robotic vehicles," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 4, pp. 744–753, 2013.

[2] P. S. Duggirala, T. T. Johnson, A. Zimmerman, and S. Mitra, “Static and dynamic analysis of timed distributed traces,” in *the 33rd IEEE Real-Time Syst. Symp. (RTSS’12)*, Dec 2012.

[3] Y. Yang, Y. Huang, X. Ma, and J. Lu, “Enabling context-awareness by predicate detection in asynchronous environments,” *IEEE Transactions on Computers*, accepted in Apr 2015.

[4] C. Baier and J.-P. Katoen, *Principles of Model Checking*. The MIT Press, 2008.

[5] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, and et al, “Spanner: Google’s globally-distributed database,” in *USENIX Conf. on Operating Systems Design and Implementation (OSDI’12)*, 2012, pp. 251–264.

[6] S. D. Stoller, “Detecting global predicates in distributed systems with clocks,” *Distrib. Comput.*, vol. 13, no. 2, pp. 85–98, 2000.

[7] A. D. Kshemkalyani and J. Cao, “Predicate detection in asynchronous pervasive environments,” *IEEE Trans. Computers*, vol. 62, no. 9, pp. 1823–1836, 2013.

[8] F. C. Gärtner, “Fundamentals of fault-tolerant distributed computing in asynchronous environments,” *ACM Comput. Surv.*, vol. 31, no. 1, pp. 1–26, Mar 1999.

[9] Y. Yang, Y. Huang, J. Cao, X. Ma, and J. Lu, “Formal specification and runtime detection of dynamic properties in asynchronous pervasive computing environments,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 8, pp. 1546–1555, Aug 2013.

[10] H. Wei, Y. Huang, J. Cao, X. Ma, and J. Lu, “Formal specification and runtime detection of temporal properties for asynchronous context,” in *IEEE Intl. Conf. on Pervasive Computing and Comm. (PerCom’12)*, 2012, pp. 30–38.

[11] MIPA - Middleware Infrastructure for Predicate detection in Asynchronous environments. http://alg-nju.github.io/mipa/.

[12] R. Cooper and K. Marzullo, “Consistent detection of global predicates,” in *ACM/ONR Workshop on Parallel and Distributed Debugging*, 1991, pp. 167–174.

[13] B. Patt-Shamir and S. Rajsbaum, “A theory of clock synchronization,” in *the Annual ACM Symp. on Theory of Computing (STOC’94)*, 1994, pp. 810–819.

[14] D. L. Mills, “Internet time synchronization: the network time protocol,” *IEEE Trans. Comm.*, vol. 39, no. 10, pp. 1482–1493, 1991.

[15] O. Babaoğlu and K. Marzullo, “Consistent global states of distributed systems: fundamental concepts and mechanisms,” *Distrib. Syst.*, pp. 55–96, 1993.

[16] R. Schwarz and F. Mattern, “Detecting causal relationships in distributed computations: In search of the holy grail,” *Distrib. Comput.*, vol. 7, no. 3, pp. 149–174, 1994.

[17] V. K. Garg and B. Waldecker, “Detection of strong unstable predicates in distributed programs,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 7, pp. 1323–1333, Dec 1996.

[18] Y. Huang, Y. Yang, J. Cao, X. Ma, X. Tao, and J. Lu, “Runtime detection of the concurrency property in asynchronous pervasive computing environments,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 4, pp. 744–750, Apr 2012.

[19] Y. Yang, Y. Huang, J. Cao, X. Ma, and J. Lu, “Design of a sliding window over distributed and asynchronous event streams,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 10, pp. 2551–2560, Oct 2014.

[20] R. Alur and D. L. Dill, “A theory of timed automata,” *Theoretical Computer Science*, vol. 126, no. 2, pp. 183–235, 1994.

[21] G. Behrmann, A. David, and K. Larsen, “A tutorial on uppaal,” in *Formal Methods for the Design of Real-Time Systems*, ser. LNCS. Springer Berlin Heidelberg, 2004, vol. 3185, pp. 200–236.

[22] A. Bauer, M. Leucker, and C. Schallhart, “Runtime verification for LTL and TLTL,” *ACM Trans. Softw. Eng. Methodol. (TOSEM)*, vol. 20, no. 4, pp. 14:1–14:64, 2011.

[23] D. Dill, “Timing assumptions and verification of finite-state concurrent systems,” in *Automatic Verification Methods for Finite State Systems*, 1989, pp. 197–212.

[24] F. Laroussinie, N. Markey, and P. Schnoebelen, “Model checking timed automata with one or two clocks,” in *Concurrency Theory (CONCUR)*, 2004, pp. 387–401.

[25] F. Wang, “Efficient verification of timed automata with BDD-like data structures,” *Intl. J. on Software Tools for Technology Transfer*, vol. 6, no. 1, pp. 77–97, 2004.

[26] R. Alur, C. Courcoubetis, and D. Dill, “Model-checking in dense real-time,” *Information and Computation*, vol. 104, no. 1, pp. 2–34, 1993.

[27] A. D. Kshemkalyani, “Temporal predicate detection using synchronized clocks,” *IEEE Trans. Comput.*, vol. 56, no. 11, pp. 1578–1584, Nov 2007.

[28] C. Xu and S. C. Cheung, “Inconsistency detection and resolution for context-aware middleware support,” in *ACM SIGSOFT Intl. Symp. on Foundations of Softw. Eng. (FSE’05)*, Sep 2005, pp. 336–345.

[29] Y. Huang, X. Ma, J. Cao, X. Tao, and J. Lu, “Concurrent event detection for asynchronous consistency checking of pervasive context,” in *IEEE Intl. Conf. on Pervasive Computing and Comm. (PerCom’09)*, Mar 2009.

[30] K. Marzullo and G. Neiger, “Detection of global state predicates,” in *Intl. Workshop on Distrib. Alg. (WDAG’91)*, 1991, pp. 254–272.

[31] J. Mayo and P. Kearns, “Global predicates in rough real time,” *IEEE Symp. on Parallel and Distrib. Processing (SPDP’95)*, 1995.

Appendix

A. *Case Study*

The case study of a realistic robot gathering scenario is conducted based on our RobotCar project[7], to demonstrate the effectiveness of PARO. We first describe the scenario and then discuss the effectiveness.

1) The Robot Gathering Scenario: In this scenario, two mobile robots are designed to move along the wall and coordinate to meet at the assembly point of a $3\ m \times 3\ m$ room (for further collaboration), as shown in Fig. 13. Each robot is equipped with four ultrasonic ranging sensors for the front, back, left, and right directions, as well as a wireless module for communication. The robots synchronize their clocks with a timing server in the same room. The actual difference ε between the clocks of the robots and the timing server is bounded by 10 ms. The robots collect the readings from the four sensors every second and label the readings with local timestamps. The robots start from two corners of the room, move at the speed of about 0.12 m/s, and adjust their routes according to the readings of the four sensors (to keep a constant distance from the wall) and the status (e.g., the location and the speed) of the other robot.

In the standard way of thinking about the computation, the system execution is regarded as a totally-ordered progression of the system state. From the view of users of the robots, the two mobile robots are expected to gather at the assembly point within 15 seconds as specified in property C_1 (in Section I). However, this synchronous model of time does not work in a partially synchronous system. Developers of the mobile robot system need to change to the notion of partially synchronous time. Then they can observe and analyze the execution of the mobile robots under the guidance of the PARO framework, as detailed below.

2) Effectiveness: Based on PARO, temporal properties with different real-time constraints can be conveniently specified and verified over the system execution trace at runtime. The property is first specified to MIPA. As the robots move, they send the timed trace (i.e., sensing data with local timestamps) to MIPA (running on a dedicated server in the room), and MIPA verifies the TCTL formulas at runtime.

We use the conjunction of two local predicates to express that the two robots arrive at the assembly point, i.e., $a = LP_1 \wedge LP_2$. The local predicate $LP_1 = (R_1.front \leq 600\ mm \wedge R_1.left \leq 400\ mm)$, indicating R_1 is at the assembly point. Similarly, the local predicate $LP_2 = (R_2.front \leq 600\ mm \wedge R_2.right \leq 400\ mm)$, indicating R_2 is at the assembly point.

As for one single timed path of the system execution, we only have to check whether the path satisfies $\Diamond^{[0,15000]}a$.[8] However, due to the intrinsic asynchrony in our scenario, we can get multiple possible timed paths of the system state. Thus we have to employ the modal operators ($\forall$ and $\exists$) in TCTL to cope with the uncertainty resulting from multiple possible system executions. Specifically, we specify a pair of TCTL formulas to MIPA:

$$\Phi_{C_1} = \forall\Diamond^{[0,15000]}a \text{ and } \Phi'_{C_1} = \exists\Diamond^{[0,15000]}a$$

[7]The RobotCar project: http://cs.nju.edu.cn/yuhuang/robotcar.htm.

[8]In this scenario, we take 1 ms as the unit of time.

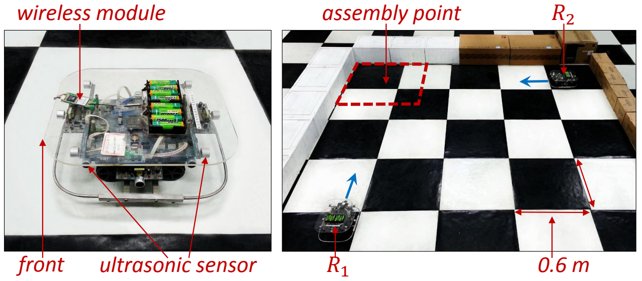

Fig. 13. The mobile robot and the scenario

There are three different situations according to the satisfaction of $\Diamond^{[0,15000]}a$ under different modalities:

1) All possible timed paths of the system state satisfy $\Diamond^{[0,15000]}a$, i.e., Φ_{C_1} is true. In this case, the real-time constraint is definitely satisfied in all possible executions;
2) Some (but not all) timed paths of the system state satisfy $\Diamond^{[0,15000]}a$, i.e., Φ_{C_1} is false and Φ'_{C_1} is true. This shows that although some paths satisfy the property, the property is possible to be violated. It indicates that the program logics of the robots have potential bugs, since the property cannot be guaranteed in all possible executions. Put it in another way, even though the property is satisfied in the current execution, it is possible that the property is violated, e.g., in the next execution;
3) None of the timed paths of the system state satisfies $\Diamond^{[0,15000]}a$, i.e., Φ'_{C_1} is false. Since the property is violated in all possible executions, we can infer that, either the specified property is beyond the capacity of the mobile robots, or there are severe flaws in the system implementation.

In our case study, we start the robots, check the formulas by MIPA, and find that Φ_{C_1} is checked false and Φ'_{C_1} is checked true. It indicates a potential violation of the property. Thus, if the user requires that the robots always gather in time, the developers should revise the program logics of the robots, e.g., by explicitly handling the uncertainty result from the asynchrony.

We further conduct two experiments to verify another two pairs of TCTL formulas. The first pair is dedicated for checking the property C_2 = "whether the robots can gather within 14,000 ms": $\Phi_{C_2} = \forall\Diamond^{[0,14000]}a$ and $\Phi'_{C_2} = \exists\Diamond^{[0,14000]}a$. Formulas Φ_{C_2} and Φ'_{C_2} are checked false, which means that the robots cannot gather within 14 s. Another pair is dedicated for checking the property C_3 = "whether the robots can gather within 16,000 ms": $\Phi_{C_3} = \forall\Diamond^{[0,16000]}a$ and $\Phi'_{C_3} = \exists\Diamond^{[0,16000]}a$. Formulas Φ_{C_3} and Φ'_{C_3} are checked true, which means that the robots can definitely gather within 16 s.

In summary, by specifying properties with different real-time requirements, and verifying the properties under different modalities based on PARO, we can gain a deep insight into the runtime behavior of the mobile robot system.

Furthermore, our 3-valued semantics is well motivated in the experiments. During the experiments, when the observed finite execution trace is not sufficient to satisfy or falsify the formulas (e.g., when the time of observation is less than 14

TABLE I. EXPERIMENTAL RESULTS OF THE CASE STUDY

n	ε (ms)	S_M (MB)	S_U (MB)	T_M (ms)	T_U (ms)
2	10	1.8	2.2	0.5	41.0

s), we are encountered with the case of "being inconclusive", while applying the classical boolean semantics may lead to false positive or false negative.

We further describe the actual performance of PARO in this case study. We evaluate the metrics S_M, S_U, T_M, and T_U defined in Section V-B. The performance of the case study is shown in Table I. We can see that the average of the total memory cost $S_M + S_U$ is less than 5 MB while the average of the total latency $T_M + T_U$ is small (less than 50 ms). This proves that PARO can verify the formulas in time and is feasible in this kind of realistic scenarios.